

# Chares are reactive

- The way we described Charm++ so far, a chare is a reactive entity:
  - If it gets this method invocation, it does this action,
  - If it gets that method invocation then it does that action
  - But what does it do?
  - In typical programs, chares have a *life-cycle*
- How to express the life-cycle of a chare in code?
  - Only when it exists
    - \* i.e. some chars may be truly reactive, and the programmer does not know the life cycle
  - But when it exists, its form is:
    - \* Computations depend on remote method invocations, and completion of other local computations
    - \* A DAG (Directed Acyclic Graph)!

# Structured Dagger (sdag)

## The *when* construct

- sdag code is written in the .ci file
- It is like a script, with a simple language
- Important: The *when* construct
  - Declare the actions to perform when a method invocation is received
  - In sequence, it acts like a blocking receive

```
entry void someMethod() {  
    when entryMethod1(parameters) { block1 }  
    when entryMethod2(parameters) { block2 }  
    block3  
};
```

Implicit  
Sequencing

# Structured Dagger

## The *serial* construct

- The ***serial*** construct
- A sequential block of C++ code in the .ci file
- The keyword ***serial*** means that the code block will be executed without interruption/preemption
- Syntax: ***serial <optionalString> /\*C++ code\*/ }***
- The ***<optionalString>*** is just a tag for performance analysis
- Serial blocks can access all members of the class they belong to

```
entry void method1(parameters) {  
    when E(a)  
    serial  
        { thisProxy.invokeMethod(10, a);  
          callSomeFunction(); }  
    ...  
};
```

```
entry void method2(parameters) {  
    ...  
    serial “setValue” {  
        value = 10;  
    }  
};
```

# Structured Dagger

## The *when* construct

```
entry void someMethod() {  
    serial { /* block1 */ }  
    when entryMethod1(parameters) serial { /* block2 */ }  
    when entryMethod2(parameters) serial { /* block3 */ }  
};
```

- Sequentially execute:
  1. /\* block1 \*/
  2. Wait for `entryMethod1` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute /\* *block2* \*/
  3. Wait for `entryMethod2` to arrive, if it has not, return control back to the Charm++ scheduler, otherwise, execute /\* *block3* \*/

# Structured Dagger

## The *when* construct

- You can combine waiting for multiple method invocations
- Execute “*code-block*” when M1 and M2 arrive
- You have access to param1, param2, param3 in the code-block

**When M1(int param1, int param2), M2(bool param3)**

{ *code block* }

# Structured Dagger

## Boilerplate

- Structured Dagger can be used in any entry method (except for a constructor)
- For any class that has Structured Dagger in it you must insert:
- The Structured Dagger macro: *[ClassName]\_SDAG\_CODE*

# Structured Dagger

## Boilerplate

The .ci file:

```
[mainchare,chare,array,..] MyFoo {  
    ...  
    entry void method(parameters) {  
        // ... structured dagger code here ...  
    };  
    ...  
}
```

The .cpp file:

```
class MyFoo : public CBase MyFoo {  
    MyFoo_SDAG_Code/* insert SDAG macro */  
public:  
    MyFoo() {}  
};
```

# Structured Dagger

## The `when` construct: refnum

- The `when` clause can wait on a certain reference number
- If a reference number is specified for a `when`, the first parameter for the `when` must be the reference number
- Semantics: the `when` will “block” until a message arrives with that reference number

```
when method1[100](int ref, bool param1)
    /* sdag block */

...
serial {
    proxy.method1(200, false); /* will not be delivered to the when */
    proxy.method1(100, true); /* will be delivered to the when */
}
```

# Structured Dagger

## The *if-then-else* construct

- The *if-then-else* construct:
  - Same as the typical C if-then-else semantics and syntax

```
if (thisIndex.x == 10) {  
    when method1[block](int ref, bool someVal) /* code block1 */  
} else {  
    when method2(int payload) serial {  
        //... some C++ code  
    }  
}
```

# Structured Dagger

## The *for* construct

- The *for* construct:

- Defines a sequenced *for* loop (like a sequential C for loop)
- Once the body for the *i*th iteration completes, the *i* + 1 iteration is started

```
for (iter = 0; iter < maxIter; ++iter) {  
    when recvLeft[iter](int num, int len, double data[len])  
        serial { computeKernel(LEFT, data); }  
    when recvRight[iter](int num, int len, double data[len])  
        serial { computeKernel(RIGHT, data); }  
}
```

- *iter* must be defined in the class as a member

```
class Foo : public CBase Foo {  
    public: int iter;  
};
```

# Structured Dagger

## The `while` construct

- The `while` construct:

- Defines a sequenced `while` loop (like a sequential C while loop)

```
while (i < numNeighbors) {  
    when recvData(int len, double data[len]) {  
        serial {  
            /* do something */  
        }  
        when method1() /* block1 */  
        when method2() /* block2 */  
    }  
    serial { i++; }  
}
```

# Structured Dagger

## The *overlap* construct

- The **overlap** construct:

- By default, Structured Dagger constructs are executed in a sequence
- **overlap** allows multiple independent constructs to execute in any order
- Any constructs in the body of an **overlap** can happen in any order
- An **overlap** finishes when all the statements in it are executed
- Syntax: **overlap { /\* sdag constructs \*/ }**

What are the possible execution sequences?

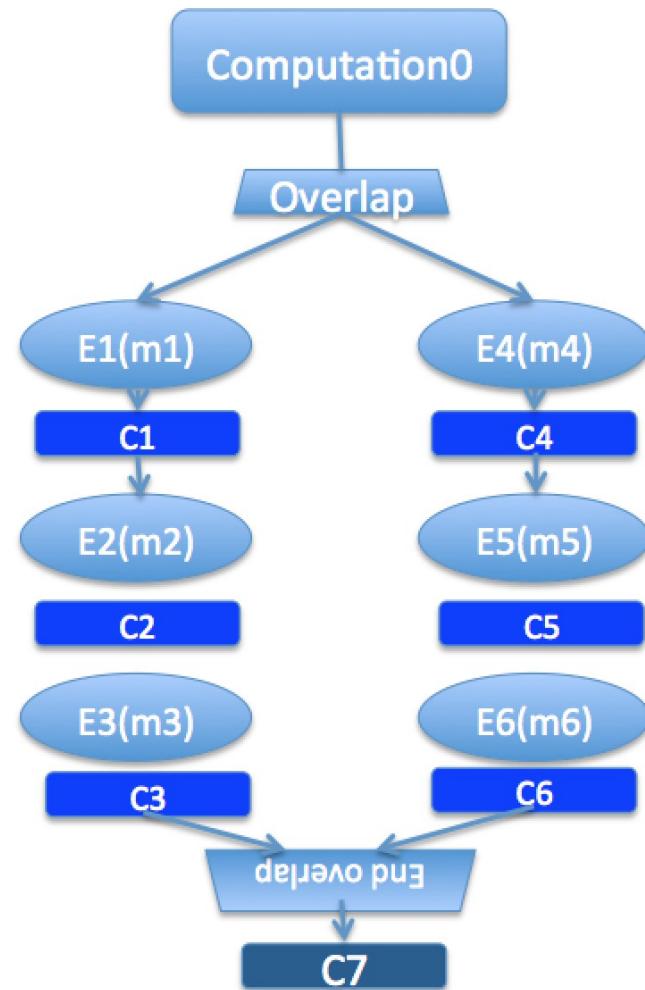
```
serial { /* block1 */ }

overlap {
    serial { /* block2 */ }
    when entryMethod1[100](int ref num, bool param1) /* block3 */
    when entryMethod2(char myChar) /* block4 */
}

serial { /* block5 */ }
```

# Illustration of a long “overlap”

- *Overlap* can be used to regain some asynchrony within a chare
  - But it is constrained
  - More disciplined programming,
  - with fewer race conditions



# Structured Dagger

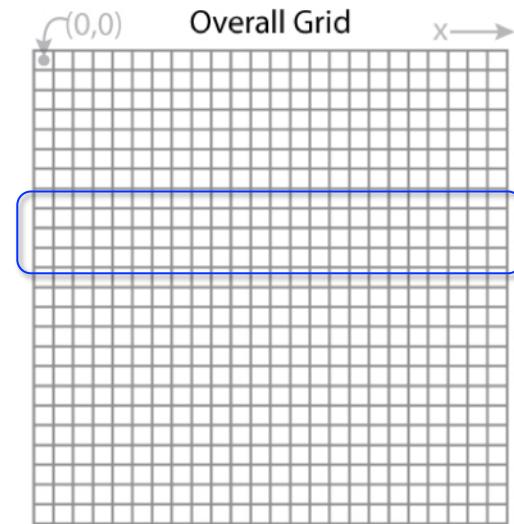
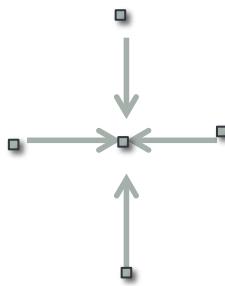
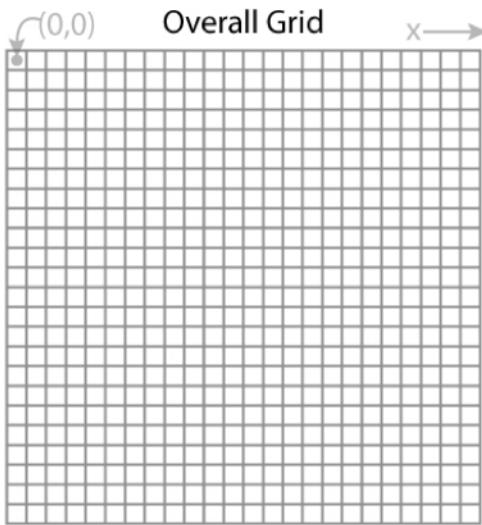
## The *forall* construct

- The *forall* construct:
  - Has “do-all” semantics: iterations may execute in any order
  - Syntax:  
`forall [<ident>] (<min> : <max>, <stride>) <body>`
  - The range from *<min>* to *<max>* is inclusive

```
forall [block] (0 : numBlocks - 1, 1) {  
    when method1[block](int ref, bool someVal) /* code block1 */  
}
```

- Assume *block* is declared in the class as *public: int block;*

# 5-point Stencil



1-D decomposition: each chare object owns a strip  
Need to exchange top and bottom boundaries



# Jacobi: .ci file

```
mainmodule jacobi1d {
    readonly CProxy Main mainProxy;
    readonly int blockDimX;
    readonly int numChares;

    mainchare Main {
        entry Main(CkArgMsg *m);
    };
    array [1D] Jacobi {
        entry Jacobi(void);
        entry void recvGhosts(int iter, int dir, int size, double gh[size]);
        entry [reductionontarget] void isConverged(bool result);
        entry void run() {
            // ... main loop (next slide) ...
        };
    };
};
```

```

while (!converged) {
    serial "send_to_neighbors" {
        iter++;           top = (thisIndex+1)%numChares; bottom = ...;
        thisProxy(top).recvGhosts(iter, BOTTOM, arrayDimY, &value[1][1]);
        thisProxy(bottom).recvGhosts(iter, TOP, arrayDimY, &value[blockDimX][1]); }

    for(imsg = 0; imsg < neighbors; imsg++)
        when recvGhosts[iter] (int iter, int dir, int size, double gh[size])
            serial "update_boundary" {
                int row = (dir == TOP) ? 0 : blockDimX+1;
                for(int j=0; j<size; j++) value[row][j+1] = gh[j]; }

    serial "do_work" {
        conv = check_and_compute(); // conv: a boolean indicating local convergence
        CkCallback cb = CkCallback(CkReductionTarget(Jacobi, isConverged), thisProxy);
        Contribute(sizeof(bool), &conv, CkReduction::logical_and, cb); }

    when isConverged(bool result) serial "check_converge" {
        converged = result; if (result && thisIndex == 0) CkExit(); }
}

```

```

while (!converged) {
    serial "send_to_neighbors" {
        iter++;           top = (thisIndex+1)%numChares; bottom = ...;
        thisProxy(top).recvGhosts(iter, BOTTOM, arrayDimY, &value[1][1]);
        thisProxy(bottom).recvGhosts(iter, TOP, arrayDimY, &value[blockDimX][1]); }

    for(imsg = 0; imsg < neighbors; imsg++)
        when recvGhosts[iter] (int iter, int dir, int size, double gh[size])
            serial "update_boundary" {
                int row = (dir == TOP) ? 0 : blockDimX+1;
                for(int j=0; j<size; j++) value[row][j+1] = gh[j]; }

    serial "do_work" {
        conv = check_and_compute(); // conv: a boolean indicating local convergence
        CkCallback cb = CkCallback(CkReductionTarget(Jacobi, isConverged), thisProxy);
        Contribute(sizeof(bool), &conv, CkReduction::logical_and, cb); }

    when isConverged(bool result) serial "check_converge" {
        converged = result; if (result && thisIndex == 0) CkExit(); }

    if (iter % LBPERIOD == 0) {serial "start_lb" { AtSync();} when ResumeFromSync() {}}
}

```